HW5: Fine Tuning the Summation Algorithm

Course:	ENEE759K/CMSC751
Title:	Fine Tuning the Summation Algorithm
Date Assigned:	April 8th, 2010
Date Due:	April 23rd, 2010 **11:59pm**
Contact:	George Caragea – george@cs.umd.edu

1 Assignment Goal

The goal of this assignment is to implement a well known PRAM algorithm and fine-tune it to obtain the an optimized implementation using the XMT platform. The algorithm used for this assignment is *Summation*, which is discussed in Section 2 of the class notes.

2 Description of the Summation Algorithm

A full description of the the Balanced Binary Tree Summation Algorithm is included in the class notes in Sections 2.1 and 2.2. For completeness, we include a WD presentation of the algorithm below.

Input: An array $A = A(1), \ldots, A(n)$ of n numbers.
The problem is to compute $A(1) + \cdots + A(n)$.

Algorithm 1 Balanced Binary Tree Summation

1: **for** $i, 1 \leq i \leq n$ **pardo**
2: $B(0,i) := A(i)$
3: **for** $h := 1$ **to** $\log n$ **do**
4: **for** $i, 1 \leq i \leq n/2^h$ **pardo**
5: $B(h,i) := B(h-1, 2i-1) + B(h-1, 2i)$
6: **end for**
7: **end for**
8: **for** $i := 1$ **pardo** output $B(\log n, 1)$

3 The XMT Software Development Environment

For this assignment, you will **not** be using the XMT FPGA Paraleap system to compile and execute your program. Instead, you are required to download and install the *XMT Software Development Environment*

(XMT SDE), freely available as part of the XMT Software Release. The XMT SDE provides a fully functional XMT programming environment.

The XMT SDE includes the XMTC Compiler, an XMT Cycle-Accurate Simulator (XMTSim) and a number of other tools and utilities, plus their documentation. The main goal of the XMT SDE is to provide a fully functional XMT Environment to anyone who is interested in experimenting and evaluating the XMT Platform, without requiring access to the XMT FPGA Paraleap computer, or purchase of expensive prototype hardware.

Important: The XMT Cycle-Accurate simulator by default is configured to emulate an XMT Configuration with **1024 parallel processors**. This is much higher than the 64 processors available in the FPGA Paraleap prototype. Keep this in mind when optimizing your implementation below.

Part of this assignment, you are required to visit the XMT Software Release web page `http://www.umiacs.umd.edu/users/vishkin/XMT/sw-release.html`. Please read this page entirely, then select the download and installation option most appropriate for your system (most Windows and Mac users will probably need to use the XMT VirtualBox option).

Note that you will need to enable Internet connectivity on the machine where you install the XMT SDE. If you plan to use the XMT VirtualBox option, make sure you follow the steps specified in documentation that enable network connectivity inside the VirtualBox.

There are a number of differences between the XMT Paraleap and the XMT SDE Environments. In particular, the commands for executing a program are different, since the simulator, and not the FPGA, is used. Please refer to the documentation included in the XMT SDE for information on the new commands and syntax. The most relevant document for these changes is the **XMT Tutorial**, which includes examples on how to use all the new commands.

4 Algorithm Fine-Tuning

The Summation algorithm above is presented using a balanced binary tree approach. In the PRAM model, this algorithm is optimal, achieving $W = O(n)$ work and $T = O(\log n)$ time.

However, because of various implementation issues, this might not be the best performing solution when implemented on actual hardware. This is mostly due to the constant factors (which are hidden in the big-Oh notation) and system overheads, including: latency of access and contention for shared resources, allocating a thread to a physical processor, starting and ending a parallel section.

Given this, several variations of the Balanced-Binary Tree algorithm can be considered as alternative implementations for the Summation algorithm. The following list describes a number of implementations that you are to experiment with:

Balanced Binary Tree Summation (BBT-S) This is the standard PRAM Summation algorithm, as presented in Section 2 above.

Balanced k-ary Tree Summation (BkBT-S) Use a balanced k-ary instead of binary tree for the Summation algorithm. In this data structure, each node has $k \geq 2$ children. k is usually a small constant (e.g. 4, 5, ...)

Serial Summation (SER-S) Serially iterate through the elements of the input array A and compute the sum.

p-Serial Summation (pSER-S) Assume that p processors are available and that n is divisible by p. Start p threads; each thread will compute the sum of a group of n/p elements. Use a different summation algorithm to add up the p sums and compute the total sum.

No-Busy-Wait Summation (NBW-S) This algorithm implements the PRAM Summation algorithm using only one spawn block. [1]

An informal description of NBW-S the algorithm is presented next. Just like in the BBT-S algorithm, the execution proceeds by traversing a binary tree from leaves towards the root, where the values of A are stored at the leaves. In addition to a numerical value, consider that each node in the tree also stores a *gatekeeper*. The gatekeeper will ensure that the sum of the node is computed after the sum for both its children is ready. The gatekeeper is implemented using an "enabling" counter, denoted $E(k)$, which is initialized to 0.

A thread i for each node at height 1 (one level above the leaves) in the binary tree is started, for a total of $n/2$ threads. After adding its two children leaves to produce $SUM(i)$, thread i will perform a "prefix-sum" operation[2] relative to $E(Parent(i))$. The prefix-sum operation will reveal to the thread if it was the first child of $Parent(i)$ to do that. If yes, the thread terminates. However, if it was the second, the thread will proceed to add the two children of $Parent(i)$. Then it will advance to its parent (i.e. $Parent(Parent(i))$) and so on. The thread which computes the sum at the root ends the computation.

In addition to these different implementations, performance can be increased further by combining two or more of these algorithms using the *Accelerating Cascades* technique. For example, one can apply the pSER-S algorithm to reduce the size of the problem to p; then switch to BkBT-S algorithm for reducing the size of the problem even further; and finally, use SER-S to finish the computation when the number of partial sums to add is below a certain threshold.

5 Assignment

5.1 Algorithm Implementations

For each of the items below, provide a source file under the *src* directory. A template file for each item is already provided at the same location.

1. Serial summation (SER-S): `ser-s.c`
2. p-Serial summation (pSer-S): `pser-s.c`. The value of p (the number of processors to use) is specified as a constant in the template file. As mentioned above, the XMT Cycle-Accurate simulator is configured to emulate an XMT System with 1024 parallel processors.
3. Balanced Binary Tree summation (BBT-S): `bbt-s.c`
4. Balanced k-ary Tree Summation (BkBT-S): `bkbt-s.c`. You are required to determine empirically the optimal value for k. This should be the value that provides best performance on the largest provided dataset (dm3). You are supposed to report the optimal value of k in the source file (see comments at the top of the template file).
5. No-Busy-Wait summation: `nbw-s.c`.
6. Best summation: `best-s-1024.c` and `best-s-64.c`. Provide here a best-effort implementation of the summation algorithm for XMT. This can use any of the algorithms provided above, as well as techniques such as accelerating cascades, to provide best possible performance. Use your own judgement, as well as the experience gained from the above implementations, to provide this implementation.

   As described below in Section 5.6, you will run your program on two simulated XMT configuration: one with 1024 TCUs and one with 64 TCUs. The optimal solution (or some parameters) might be

[1] For your background, this algorithm is based on the no-busy-wait balanced tree paradigm introduced in [1].

[2] Hint: You can use the XMT prefix-sum primitives to implement this.

different depending on these two configurations. Provide the best solution for the 1024 TCUs in the `best-s-1024.c` file and the best solution for the 64 TCUs in the `best-s-64.c` file.

Important: Provide a text file called `best-s.txt` with a brief description of your best-s algorithm.

The performance of each of your implementations will be graded indepently, by comparing with the reference cycle counts, included in Section 6.

5.2 Installing XMT Software Development Environment

Download and install the XMT Software Development Environment as described in Section 3.

5.3 Setting Up the Environment

The header files and the binary files can be downloaded from the class assignment web page:
`http://www.cs.umd.edu/~george/xmt/enee759k_sp10/assignments/summation.tgz`
To get the data files, log in to the machine containing the XMT SDE (e.g. the XMT VirtualBox), and run the following commands:

```
$ wget --user=class --password=parallel \
  http://www.cs.umd.edu/~george/xmt/enee759k_sp10/assignments/summation.tgz
$ tar xzvf summation.tgz
```

This will create the directory *summation* with following folders: *data*, and *src*. Data files are available in the data directory. Edit the *c* files in *src*.

5.4 Data Format for Summation Algorithm

#define n	The number of elements in array *A*.
#define logn	Logarithm of *n* in base 2.
int A[n]	The input array A.
int sum	Output: the total sum.

You can declare any number of global arrays and variables in your program as needed. $\log n$ is an integer. The result (sum of elements in A) should be stored in `sum` (the grading scripts will check for the value of this variable).

5.5 Data Sets for All Summation Implementations

You should run your program using the following data sets including the header file with the compiler option *-include*. You can also use the *#include* directive inside the XMTC file, however remove these directives before submitting your code. Alternatively, you can use the Makefile system using the commands described in Section 5.7 below.

Data Set	**n**	**Header File**	**Binary file**
dm1	128	data/dm1/summation.h	data/dm1/summation.xbo
dm2	64k	data/dm2/summation.h	data/dm2/summation.xbo
dm3	1M	data/dm3/summation.h	data/dm3/summation.xbo

Each data directory includes a correctout.txt file that contains the correct output for the matching input.

Note that the XMT Cycle-Accurate simulator is significantly slower than the Paraleap FPGA prototype when executing XMTC programs (about 10,000 times slower by our measurements). Therefore, simulation time can be lengthy for larger datasets, or inefficient programs. For the largest dataset (dm3), our reference solutions simulate in approximately 10-15 minutes.

5.6 Simulated Configurations

One of the main advantages of the XMT Simulator is that it can be configured to simulate a wide range of XMT configurations and experiment with different hardware parameters. This is very difficult and expensive to do using real hardware.

For the current assignment, you are to experiment with two simulator configurations:

- A 64-TCU XMT configuration, resembling the Paraleap FPGA prototype.
- A 1024-TCU XMT configuration, which represents a more forward-looking architecture, but which is actually feasible with current manufacturing technology.

To switch between configurations, you need to pass the parameter `-conf fpga` or `-conf 1024` to the simulator command line. Alternatively, as it will be explained below, you can use the provided makefile to switch.

5.7 Compiling and executing via the Makefile system

You can use the provided makefile system to compile and run your programs. For the smallest data set (dm1) serial summation program is run as follows (example shows using fpga 64-TCU configuration):

```
> make run INPUT=ser-s.c DATA=dm1 CONF=fpga
```

This command will compile and run the ser-s.c program with the dm1 data set. For other programs and data sets, change the name of the input file and the data set. For the 1024-TCU configuration, change to `CONF=1024`.

If you need to just compile the input file (no run):

```
> make compile INPUT=ser-s.c DATA=dm1
```

You can get help on available commands with

```
> make help
```

Note that, you can still use the `xmtcc` and `xmtsim` commands as described by the documentation.

5.8 Obtaining Performance Information

When executed with the "-cycle" option (which is the default method when using the `make run` method above, the XMT Cycle-Accurate simulator reports the number of cycles of the execution:

```
Total Execution Time = 23429
```

This number gives the running time of the program. This is what you are required to optimize as part of the fine-tuning assignment.

In addition to cycle-counts, the XMTSim can also provide the total number of instructions it executed on all the processors (as well as other, more detailed statistics about the execution). These statistics are printed when using the "-count" option (in addition to "-cycle"). The number of instructions is reported:

```
Number of instructions: 4194335
```

This number effectively measures the *Work* of the algorithm. Note that you are **not** required to optimize for *Work*, however, it might provide you with useful information as you are fine-tuning your implementations. Detailed execution statistics are displayed by default if using `make run_stats` command instead of `make run`.

6 Reference Performance Results

The reference cycle counts and speedups vs. serial execution, using the dm2 dataset are presented in Table 1. Extracredit will be awarded for implementations that outperform these reference cycle counts.

Program	Cycle counts
ser-s.c	68,157,670
bbt-s.c	172,456
bkbt-s.c	107,138
pser-s.c	106,323
nbw-s.c	467,339
best-s-1024.c	35,740
best-s-64.c	358,170

Table 1: Reference cycle counts and speed-ups for the dm3 dataset.

7 Grading Criteria and Submission

In this assignment, you will be graded on the correctness of your programs (ser-s.c, pser-s.c, bbt-s.c, bkbt-s.c, nbw-s.c, best-s-1024.c and best-s-64.c) and the performance of each implementation in terms of completion time. Performance of your program will be compared against the performance of our reference implementations, provide in Section 6. Don't forget that you are also supposed to report the best value obtained for the *k* factor for the BkBT-S in `bkbt-s.c`.

Please remove any `printf` statements that you may have placed for debugging purposes from your code as they will affect the performance and possibly break the automated grading script. Once you have all the source files under the *src* directory you can check the correctness of your programs[3]:

```
> make testall
```

check the validity of your submission:

```
> make submitcheck
```

and submit:

```
> make submit
```

Note that you will be asked to enter the class username and password when doing the submission. These are the same ones that you used to login to the Paraleap server.

[3] `make testall` command runs all possible combinations of programs and data files. This is likely to take a long time on the XMT Simulator, therefore use with care.

References

[1] Uzi Vishkin. A no-busy-wait balanced tree parallel algorithmic paradigm. In *SPAA '00: Proceedings of the twelfth annual ACM symposium on Parallel algorithms and architectures*, pages 147–155, New York, NY, USA, 2000. ACM.